THE UNIVERSITY OF
WARWICK

Original citation:
Beynon, Meurig and Sun, P. H. (1998) Interactive situation models for program comprehension. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-352

Permanent WRAP url:
http://wrap.warwick.ac.uk/61065

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:
The version presented in WRAP is the published version or, version of record, and may be cited as it appears here.For more information, please contact the WRAP Team at:
publications@warwick.ac.uk

warwickpublicationswrap
highlight your research

http://wrap.warwick.ac.uk/

Research Report 352

Interactive Situation Models for Program Comprehension

Meurig Beynon, Pi-Hwa Sun

RR352

The complexity of the interactions between programmable components and human agents in modern computing applications motivates new approaches to program comprehension. Understanding the role of programs within a reactive system, for instance, involves not only input-output transformations, but also communication and stimulus-response issues. This paper examines the prospects for constructing novel computer-based **interactive situation models** to assist program comprehension. Such a model provides an environment within which the human interpreter can explore the data relationships and patterns of behaviour generated by a computer program with particular reference to its external real-world semantics. The proposed method of construction exploits principles based upon observation, agency and dependency ("Empirical Modelling") that have been developed at the University of Warwick. Connections between these principles and previous work on program comprehension are discussed. Some experimental studies in applying interactive situation models are reviewed, and conclusions about their potential future role in program comprehension and generation are drawn.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

October 1998

Interactive Situation Models for Program Comprehension

Meurig Beynon, Pi-Hwa Sun
Department of Computer Science, University of Warwick, Coventry CV4 7AL

Abstract

The complexity of the interactions between programmable components and human agents in modern computing applications motivates new approaches to program comprehension. Understanding the role of programs within a reactive system, for instance, involves not only input-output transformations, but also communication and stimulus-response issues. This paper examines the prospects for constructing novel computer-based ***interactive situation models*** *to assist program comprehension. Such a model provides an environment within which the human interpreter can explore the data relationships and patterns of behaviour generated by a computer program with particular reference to its external real-world semantics. The proposed method of construction exploits principles based upon observation, agency and dependency ("Empirical Modelling") that have been developed at the University of Warwick. Connections between these principles and previous work on program comprehension are discussed. Some experimental studies in applying interactive situation models are reviewed, and conclusions about their potential future role in program comprehension and generation are drawn.*

1. Introduction

A central problem in program comprehension is that of inferring dynamic behaviour from static texts. The texts to be consulted include the program code and possibly other sources such as requirements specification documents. The behaviour to be inferred concerns both the execution of the program in machine-oriented terms (how data is processed at some appropriate level of abstraction), and the interpretation of the program execution in real-world terms (what input-output and stimulus-response patterns are generated by the executing program, and what function these serve in the real-world application). For many practical purposes, it is not enough simply to be able to give a plausible behavioural account of a program; in maintenance, or adaptation to new requirements, it is essential to be able to identify in precise detail how the behaviour is related to program structure.

The task of the human interpreter of a program can usefully be regarded as dual to programming. R Brooks [12], for instance, views program comprehension as the reconstruction of the domain knowledge used by the initial developer. Good programming practice involves transforming a conception of the system into programs for its components; comprehension involves converting programs for components into a conception of the system. The quality of the program development process is significant in this context; the role of effective development techniques is to relate program construction to conception of the system. The fact that the program performs in a certain way is not the most important issue in program comprehension (in many contexts, it may be possible to observe the program execution directly), but the way this performance has been contrived. Viewed in this way, program comprehension is intimately related to fundamental issues in program development, such as:

- how does the programmer conceive the system?
- how can this conception be best represented?
- how does this conception inform the program construction?

As modern computing applications continue to become more sophisticated, the interactions between computer, human agents, interfacing devices and other electronic devices such as sensors and actuators are ever more significant. In archetypal sequential interactions between user and computer, the essence of program development and comprehension may lie in identifying and correlating functional relationships at many levels of detail, but such abstractions cannot do justice to the mental representations that are required to address state-of-the-art computer applications. The need for new concepts and address modern software

development has been discussed at length by Fred Brooks [11] and Harel [16]. Reactive systems, characterised (cf. Pnueli [20]) by embedded components, and concurrent real-time interaction, pose particular challenges. In such systems, it is not possible to abstract the role of a program from its context. Even very simple functional components can generate complex behaviour through concurrent interaction.

This paper examines the potential for applying Empirical Modelling (EM) principles, techniques and tools, as developed at the University of Warwick, to program comprehension. Such application is suggested by the fact that EM has already been used to construct models that give particular support to the early stages of reactive system development [3,8,10], and that are explanatory, in that they express a causal account of a concurrent system in terms of agents, observables and dependencies [6]. Background information on the principles and tools applied, and the models constructed, can be found at the Empirical Modelling website:

http://www.dcs.warwick.ac.uk/pub/research/modelling.

The remainder of the paper is organised in two main sections. Section 2 is a general discussion of the principles behind the application of EM to constructing interactive situation models. Section 3 outlines two illustrative examples. Further issues and future research directions are examined in the concluding section.

2. Empirical Modelling Principles and Program Comprehension

2.1. Interactive Situation Models

Previous experience has demonstrated that EM can be used to construct complex and open-ended models of concurrent systems. The character of these models is unusual. Rather than capturing system behaviour within a closed world, in the manner of a logical or mathematical model, they are particularly well-suited to representing specific real-world states. In effect, the computer serves as a cognitive artefact for the systems analyst, supplying metaphorical representations for the system state in which conceivable state changes are constrained only by the presumptions the analyst makes about plausible agency and dependency in the world.

Pennington introduced the concept of a *situation model* for program comprehension [19] by analogy with Kintsch and van Dijk's theory of text comprehension [17]. In [17], text comprehension is viewed as developing a model of the situation decribed in a given text from an associated *textbase,* comprising "a surface memory of the text, a microstructure of interrelations between text propositions, and a macrostructure that organises the text representation". In Pennington's model of program comprehension, the procedural relationships implicit in the program structure play the role of the textbase. The functional relationships between real-world objects that are reflected in the program then supply a situation model analogous to the model of the situation which the text describes.

This paper proposes that EM principles can be used to construct a model that serves the same role as Pennington's situation model, but has a different character. A situation is represented to the modeller as a computer-generated environment for exploration. The current state of the computer model represents a particular state of the system. There are many possible scenarios for transition to other states. The modeller can explore these scenarios interactively through directly redefining observables and dependencies. The modeller can also choose to introduce autonomous agents into the environment embodied in the model in order to test hypothesis about the mechanisms that are operative.

There are several reasons to suppose that there are advantages in an interactive situation model:

The role of interactive tools for system development: In complex interactions within a reactive system, communication of state is very significant (e.g. Deutsch's concern for stimulus-response patterns between agents [14]). Harel [16] contends that visual formalisms are needed to apprehend the semantics of comple: systems. Interaction is also an important feature of practical tools for debugging.

The importance of empirical elements in an engineering context: Reliable knowledge about the interactions between agents in a system is an essential prerequisite to programming its components. An engineer wishing to explain a product with a view to maintenance or modification typically has to make use of artefacts to share the knowledge that informed the product development and the experimental activity that led to particular design decisions and structural features.

The pragmatic nature of program comprehension: Research by Good and Brna [15] and others indicates that program comprehension can have many different complementary interpretations. An interactive situation model that can be subjective and is open to exploration, extension and revision is better suited as a program comprehension model than a document such as a program summary.

2.2. EM and the Programmer's Conception of a System

Constructing models to represent the interactions between the agents in a reactive system has been a central focus for research in the EM project [3,8,10]. In this context, an agent refers broadly to any component of the system that can be responsible for changes of state: this may be (for example) a computer, a sensory device, an actuator, a clock, a switch or a human agent. EM is most directly relevant to understanding the interactions in a complex system prior to explicitly constructing or programming these components. The essential idea behind EM is to construct a computer representation in which system state is represented metaphorically (typically through a visual representation), and - by default - the simulation of system behaviour is automated only to a limited degree. In effect, the current state of the computer model, as apprehended by the modeller, represents a particular state of the system (as far as this has yet been conceived or observed), and the mode of interaction with the model resembles the interaction between an experimenter and her environment. Autonomous changes of state within the computer model occur only under the discretionary control of the modeller. (A useful parallel may be drawn with interaction between a user and a spreadsheet, where semantically interesting changes of state typically occur only as a result of user actions, and the current state of the spreadsheet supplies a context for open-ended *what-if?* experiments.)

EM supplies a framework for concurrent systems conception in which the basic abstractions are *observables*, *dependencies between observables*, and *agents* that act through changing observables and dependencies. It makes use of a special-purpose interpreter in which observables are represented by variables, dependencies by scripts of definitions ("definitive scripts") resembling the definitions of cells behind a spreadsheet, and agent actions can be represented via triggered procedures. EM constructs a model that reflects the modeller's perspective on agency, synchronisation and causality within the system, subject to pragmatic judgements based on the intended application of the model. The identification of observables, dependencies and agents is arguably a process that underlies all system construction, whatever the nature of the programming activity and paradigm is used, even if this process is not explicitly addressed. The main theme of this paper is that program comprehension is assisted by trying to interpret programs with reference to these fundamental abstractions.

Research into applying EM principles to software development for reactive systems is still in its preliminary stages. EM techniques and tools are not yet sufficiently mature to address large programs, but there is much relevant work on EM to supply evidence of potential for entire system development. Several examples of model construction using EM principles have been described in previous papers [3,4,10]. These include a vehicle cruise controller, a billiards game simulation, a digital watch and statechart simulation, and a train simulation. Some research has been done into semi-automatic translation from such models to conventional programs and simulations. This includes the LSD Engine, developed by Adzhiev and Sarkisov at the Moscow Engineering Physics Institute, that uses EM principles to generate C++ programs semi-automatically.

In the context of this paper, the significant issue is how EM shapes the conception of a reactive system and

thereby provides a framework within which to address program comprehension. It would be premature to study program comprehension in realistic reactive systems at this stage, but there are demonstrable benefits in adopting a systems view even in simpler contexts. By way of illustration, the examples introduced below indicate how EM techniques can be used to give insight into programs for sequential user-computer interaction. For instance, mouse actions by the user can be usefully interpreted as invocations of agents in a manner that gives greater prominence to the user's role, and to issues relating to non-functional requirements.

2.3. Program Comprehension from an EM Perspective

Several scenarios can apply to program comprehension. It is typically reasonable to suppose that the human interpreter has some informal knowledge of the intended context and mode of use of a program. It may also be possible to execute the program, and apply debugging tools to analyse its execution. There are a variety of ways in which EM techniques can be applied under these conditions.

The description that follows presumes that the comprehension task is to understand how the components of a reactive system have been programmed to operate concurrently. This entails conventional program comprehension for the software components in the system. It must also take account of other more unfamiliar forms of agent programming, such as the recommended protocols for the human agents in the system, and the thresholds set for sensor activation etc. The illustrative examples to be introduced below make it plausible that EM can be applied in this context, and indicate some of the potential benefits that can be gained even for conventional program comprehension.

The aim of the program comprehension task is to construct an interactive situation model that (through the appropriate use of metaphors) reflects the functional relationships and stimulus-response characteristics of the components of the system. In constructing this model, the interpreter can draw upon domain knowledge to elaborate the model top-down (somewhat in the spirit of Brooks' comprehension model [12]), and upon inspection of program code to refine the model bottom-up (in the spirit of Pennington's comprehension model [19]). The use of EM is also well-oriented for Brooks' theory that hypotheses are the sole drivers of cognition; the concept that the evolving model embodies a working hypothesis about how the system functions is entirely consistent with our previous practice in modelling concurrent systems [8]. There are also precedents for integrating information drawn from top-down and bottom-up developments within a single model [4,10].

The relevant domain knowledge in this context takes the form of understanding of causal relationships and agency within the reactive system. This addressed such issues as: what are the agents in the system to whom state-changes are attributed? to what stimuli do these agents respond? what observables represent their 'perception' of system state? what actions can they perform to change observables? how are relevant observables within the system synchronised in system behaviour? what dependencies between observables pertain when actions are performed? Common-sense knowledge may be sufficient to account in general terms for the interaction in a reactive system, but more specialist insight becomes essential in points of detail. This issue is highlighted by the problems that a 19th century spectator would have in accounting for the responses of a radio-controlled vehicle.

The interactive situation model, as developed from a top-down approach, is necessarily framed in terms of real-world observables, agents and dependencies. The domain knowledge it captures includes information about what agents are present in the system, what observables they are presumed to respond to, how their action upon other agents is mediated via observables, and what dependencies and invariant relationships pertain amongst observables. To a limited degree, this model can also support experiments intended to identify the ways in which changes to observables are synchronised in communication. More precise knowledge of the mechanisms that are used by agents to perform their roles is needed for a complete understanding of how the system components have been programmed.

A top-down development leads to an interactive situation model that captures the developer's primary concept of a reactive system. Such a model will be common to many different developers, since it is concerned only with observables, agency and dependency without reference to the mechanisms for implementation. It provides a target for an analogue of Pennington's bottom-up program comprehension process, where the program text is consulted to gain insight into a program at progressively higher levels of abstraction.

Pennington classifies the products of inspection of program code in two ways, according to information type, and level of detail [19]. Her classification includes the following information types:

- control flow - information about the sequence of events occurring in the program;
- data flow - information regarding transformations of objects occurring during the program, including data dependencies and data structure information;
- state - values of variables, and how these are synchronised in change;
- operation - how particular lines of code can be interpreted as acting on state;
- function - high-level information about the overall goal of the program,

each represented at three levels of detail:

- detailed (referring to program operations and variables);
- program (referring to a program's procedural blocks);
- domain (referring to real-world objects).

From the perspective afforded by top-down development of the ISM, program comprehension is focused on identifying how a program defines the computer's role as an agent. In practical terms, when carrying out a bottom-up analysis of code, this involves:

- reconstructing the states in which the computer characteristically interacts with other agents;
- identifying the stimuli to which the computer responds and which it generates;
- assembling primitive actions into semantically meaningful transitions;
- identifying which (families of) variables represent observables;
- identifying the semantically significant dependencies between observables.

The essence of this process is cross-referencing between different levels of detail so as to reconstruct domain knowledge that is specifically associated with EM.

The particular computational abstraction that is most relevant in this connection is *definition* of the kind that has been introduced in this paper. The use of scripts of definitions to represent state and of new definitions or redefinitions (possibly executed simultaneously) to represent transitions characterises the form of computer programming most closely related to EM, viz. *definitive (definition-based) programming* [2]. In definitive programming, the computation that is associated with maintaining dependency within a script is uninterpreted; only new definitions and redefinitions are associated with externally significant actions on the part of an agent. Dependency maintenance represents activity that is both invisible to other agents and indivisible, in the sense that it cannot be interrupted through the intervention of an external action. In analysing a conventional program, the challenge is to use information about control flow, data flow, state and operation to distinguish between code that maintains the integrity of internal state from code that effects externally significant transitions.

Many features of existing programming paradigms address these issues to some extent, and offer clues to the bottom-up reconstruction process. In general, the use of data structures and assertions is concerned with integrity of state; object-orientation with maintaining local integrity of state, and the empirical validation of local behaviours; functional abstractions with suppressing hidden state. Other programming features can be related to agency and observables in a more direct manner. For instance, descriptive identifiers help to distinguish the externally meaningful observables, whilst event-driven and rule-based paradigms can be directly associated with agency. The most problematic aspect of making no explicit distinction between invisibly maintaining state and visibly performing transitions is that many computational abstractions can

be equally effective in either role. For instance, rule-based computation is used both as a device for updating state to reflect a dependency, and as a convenient way to specify behaviours. The invocation of methods may likewise be associated with communication between agents in the real-world domain or with purely private maintenance of state within a program object.

3. Applying Empirical Modelling to Interactive Situation Models for Program Comprehension

This section illustrates ideas that are representative of how EM can potentially be used to build ISMs for program comprehension in a reactive systems context. Example 1 emphasises top-down development of an ISM, whilst Example 2 examines issues in bottom-up development from an object-oriented program.

Example 1. Top-down Development of an Interactive Situation Model

Figure 1 depicts an embellished interface to a simple program, called *jugs*, that was first developed for educational use in schools (cf [4]). The intended functionality of the program is clear from the menu buttons: the objective for the pupil is to realise the specified target quantity of liquid in a jug by appropriately filling and emptying jugs and pouring from one jug to another. The *jugs* program is a simple simulation with a disguised mathematical significance: the capacities of the jugs are integers, and the operations that have to be performed in order to achieve the specified target can be interpreted as steps in Euclid's subtraction algorithm for computing the greatest common divisor of the capacities of the jugs.

In constructing an ISM for *jugs* in a top-down, it is only necessary to identify the observables that are significant in the interaction between the user and the program. These include the parameters whose status is explicitly represented to the user through the interface: the capacities of the jugs (capA, capB), their contents (contA, contB), the target, the status of the interaction (e.g. is the program awaiting input from the user? has the target been successfully achieved?), and the availability of menu options (e.g. is it possible to fill jug A?). They also include the triggers that initiate activity in the program, which can be represented by the selection of a value for an observable *input*, where input takes on one of five values depending upon which menu button is selected. The identification of the observables that mediate user-computer interaction in this way is associated with the shift in perspective that regards user and computer as agents within a simple reactive system.

It is not necessary to consult program code to be able to infer the primitive agency and dependency between the above observables. It is self-evident that in any *jugs* program, there should be dependencies between observables such that (for example) the menu option *fillA* is available if and only if jug A is not full, and that this is the case precisely when $0 < contA < capA$. The semi-realistic manner in which the capacity and content if the jugs is represented to the user indicates that interaction is mediated by yet another observable *fullA* - viz. whether jugA is full. The criterion by which a feature of the current state of the system is viewed as an observable to the user is that it should be possible to apprehend its status directly. The same criterion can be applied to other agents within the system, subject to introspecting about how it is that we conceive that a system operates. For instance, it is reasonable to presume that the selection of a menu button is instantaneously registered by the computer.

Where agency is concerned, it is clear that the state changes associated with pouring liquid from one jug to another are carried out by the computer, and that the context in which this function is performed is such that it can be regarded as one of several independent roles that the computer performs in response to user input. That is to say, pouring liquid between jugs is an operation initiated by the user that can be construed as invoking a *pouring* agent whose sole purpose is to simulate pouring and return control to the user. For this reason, it is appropriate to take an agent-oriented view of the computer's role, thereby elaborating the conception of the user and computer as a reactive system in a way that assists comprehension.

The jugs model illustrates the principal characteristics of an Empirical Modelling ISM. Such a model can

be constructed in a special-purpose environment supplied by the tkeden interpreter [1]. Variables such as capA, contA, capB, fullA, availFillA, input etc are used to represent the observables the mediate between agents. Definitions of variables are used to express dependencies between these observables. New definitions and redefinitions or variables (freely available to the modeller in the role of super-agent) are used to represent changes of state within the model. Such changes can be associated with the experimental process involved in the construction of the model, but can also represent typically actions of agents, such as the user or the *pouring* agent in *jugs*. These definitive (definition-based) aspects of the tkeden interpreter include means to design line drawings and window layouts for the metaphorical representation of system state. Their versatility is also enhanced by conventional procedural constructs, and event-driven procedural actions. These enable the modeller to introduce rich functional dependencies into dependency relations, and to simulate circumscribed behaviour of agents internal to the model.

Fuller details of a *jugs* model can be found in [4]. The following simple fragments from the model illustrate the essential characteristics.

The definitive script that represents the observables and dependencies has the general form:

contentA = ...
capA = ...
fullA = *contentA*==*capA*
avail_option_fillA = *not fullA*
...

Notice that this script can incorporate more explicit information about how interaction between user and computer is mediated. For instance, the definition establishes a colour convention for menu availability:

colour_button_fillA = *if avail_option_fillA then red else green*
....

Dependencies of this nature make the ISM more program-specific. They can reflect the particular capabilities of the computer on which the *jugs* program is implemented (e.g. presuming a colour display), and of the user (e.g. the user is not colour blind). Though all ISMs for a particular system will include representations of the fundamental observables, agents and dependencies, agent-specific features will have to be introduced into the model to support deeper comprehension of a particular implementation. Such features will have to be inferred from knowledge of the implementation architecture, and of the primitives used in the program code. For instance, as Figure1 illustrates, the visualisation of the jugs can be developed using simple line drawing primitives (e.g. point and line drawing), using window and text layout primitives, or generated as a textual output. In working with these display architectures, three different types of definitive script are appropriate. All three have been used simultaneously in generating Figure 1.

The modeller can interact with the *jugs* ISM in the role of a super-agent, by directly redefining the values of key parameters, for example. By way of illustration, the redefinition

capA = 2 * | *capB* |

assigns the capacity of A to twice the current capacity of B (the construct '| ...|' is used to refer to the current value of an observable), whilst the redefinition

contentA = *target*

can be used to simulate successful completion of the user's task. Activity of this kind plays an important part both in the model construction and the program comprehension process. It enables the modeller to

determine the expected values of parameters within the program when the basic dependencies are being respected. In this respect, the ISM resembles an environment in which semantically significant assertions about relationships beween programs variables are being animated.

A super-agent mode of interaction with the ISM is in some respects analogous to program debugging. It permits state-changes that can be informative in program comprehension, but are beyond the functionality of the program. For *jugs*, the valid interactions are defined by a choice of input menu, represented by a redefinition of a variable *input* (=*fillA, fillB, emptyA, emptyB, pour*), as in the user interaction:

if avail_option_fill then input = pour

Although the modeller can simulate the interactions of the user and other agents directly, it is often convenient to automate an agent role. For instance, the *pouring* agent can be explicitly modelled within the *jugs* ISM by an event-driven action. A simple extract from the protocol for this agent is:

if input==pour and not emptyA and not fullB then
{input=pourAB; contentB = | contentA+contentB |-contentA}
if input==pourAB and not emptyA and not fullB then contentA = |contentA |-1

More background on the *jugs* model can be found in other references [1,4]. Their emphasis is not upon use of the *jugs* model as an ISM, but as a medium for software development. For example, it is possible to derive a conventional procedural program from the jugs model by semi-automatic translation.

Example 2. Developing an Interactive Situation Model for a Simple Object-Oriented Program

Studying a particular program with a view to deriving an interaction situation model helps to clarify the principles discussed in this paper, and to expose the significance of the EM process. It also illuminates Brooks' thesis that program comprehension entails a reconstruction of domain knowledge used by the program developer. This section introduces an ISM to assist the comprehension of a simple object-oriented program for managing a mailbox taken from a standard textbook on JAVA [13] (cf. Figure 2).

For the JAVA programmer who develops the mailbox, the system concept is informed at the highest level of abstraction by a requirements specification. This might take the form: "A mailbox keeps up to 10 messages that can be read on request. The number of messages currently in the mailbox is also displayed on request. New messages can be introduced into the mailbox if space permits, and messages once read are discarded. When mail is read, messages are presented in FIFO order." In working from this specification, the programmer has to conceive the objects that are to be used to represent the state of the mail system, and the methods that are to be invoked to manipulate this state. For this purpose, a state-change in the real world has to be expressed as corporate activity on a family of objects.

The following listing specifies the class mailbox:

```
class mailbox
{
    public message remove()
    {   if (nmsg == ) return null;
        Message r = messages[out];
        nmsg--;
        out = (out+1) % MAXMSG;
        return r;
    }
```

```
    public void insert(Message m)
    {       if (nmsg == MAXMSG) return;
            messages[in] = m;
            nmsg++;
            in = (in+1) %MAXMSG;
    }

    public String status()
    {       if (nmsg==0) return "Mailbox empty";
            else if  (nmsg==1) return "1 message";
            else if (nmsg < MAXMSG) return nmsg + "messages";
            else return "Mailbox full";
    }

    private final in MAXMSG = 10;
    private int in = 0;
    private int out = 0;
    private int nmsg = 0;
    private Message[] messages = new Message[MAXMSG];
}
```

In arriving at this specification, the developer has decided to use a queue data type to store messages so that they can be displayed in FIFO order. The queue is implemented via an array, and the variables *in* and *out* are pointers to the first and last elements of the queue. Knowledge of this representation supplies the bridge between the requirements specification and the program code.

At the maintenance stage, a programmer will have to recognise the mapping from real-world interpretation to program code. The concept of "displaying messages by FIFO" has to be identified with sequences of operations on particular variables, such as the pointers *in* and *out* and the array *messages*. More generally, the current state of the executing program is what results from the cumulative effect of method invocations upon the collection of objects that is initially instantiated. This further complicates cross-referencing between the program level and the domain level of detail.

Once the queue representation has been understood, it becomes possible to construct an explicit model of the states of the mailbox that is based on essentially the same observables. The definitive script below, written in a variant of the *tkeden* notation that includes special-purpose windows of type TEXTBOX for text entry and display, expresses the current state of the mailbox, and serves as an ISM for the JAVA program above. In this script, an '=' system indicates an assignment, and an 'is' a definition. In the environment of the script, it is possible for the modeller to take actions outside the scope of the JAVA program. For instance, the modeller can act in a super-agent role to simulate the corruption of messages through an operating system failure. Actions of this kind can be useful when exploring the intended and potential behaviours of the associated program.

```
Maibox is [message1, message2, message3, ...., message10];

// define the maximum number of messages that can be stored to be the size of the mailbox
MAXMSG is Mailbox#;

// initialise sender and content as lists comprising MAXMSG strings
sender = <list of MAXMSG items, each the empty string>;
content = <list of MAXMSG items, each the empty string>;
```

```
message1 is [sender[1], content[1]];
message2 is [sender[2], content[2]];
message3 is [sender[3], content[3]];
....
message10 is [sender[10], content[10]];

in is (totalin % MAXMSG) + 1
out is (totalout % MAXMSG) + 1;
nmsg is totalin - totalout;
totalin = 0;
totalout = 0;

proc Text_click {
        if (nmsg>MAXMSG) return;
        sender[in] = fromText_getText();
        content[in] = msgtext_getText();
        totalin = totalin+1;
        fromText_setText("");
        msgText_settext("");
}

proc Play_click {
        if (nmsg==0) return;
        fromText_setText(sender[out]);
        msgText_setText(content[out]);
        totalout = totalout+1;
}
```

In this ISM, the actions Text_click and Play_click, respectively associated with supplying a new message and reading a message from the Mailbox, are closely modelled on the original JAVA program. The definitions of *in* and *out* are derived by interpreting the queue implementation for stored messages. The descriptive state-oriented rather than behaviour-oriented nature of the ISM model establishes a direct link between the model and an observation-and-agent-oriented conception of the system.

4. Concluding remarks

This paper has focused on program comprehension at the domain level. Similar principles can be applied when examining programs at higher levels of detail. A recent application of EM principles to the exposition and study of heapsort [9] demonstrates the potential for constructing interactive situation models to support programming activity at much lower levels of abstraction. In this context, the relevant observables are the semantically significant features of the data structures (such as the order relations between values at adjacent nodes of a tree, and abstract conditions such as whether the heap condition is satisfied at a node). The problems of program comprehension can also be viewed as seamlessly connected with the comprehension problems that arise in mathematical and scientific visualisation. It is natural to seek an extension to the ISM for the **jugs** program that clarifies the number-theoretic significance of the simulation activity, for example. There have been several previous applications of EM to visualisation problems of this kind [7]. The range of possible ways in which programs can be interpreted recalls the distinction made by Smith [21] between the operational semantics of programs, as understood by computer scientists, and a program's meaning in relation to its external context - "the semantics of the semantics of programs" (cf [5]).

The potential applications of EM to requirements [8], together with research that has been done on automatic generation of conventional programs from EM models of concurrent systems point to a futuristic

scenario in which an interactive situation model is supplied with a program. This could be useful as a extensible basis for program comprehension, as a way of linking requirements specification and validation, and as a knowledge base from which to derive program variants.

Previous experience of developing models using EM also points to significant issues for future research:

Problems of scale: The size of models that we have been able to construct with our present tools is restricted to at most a few thousand definitions. Large models can be hard to manage because they are less structured than good conventional programs, include many dependency links and do not have information hiding. Some possible research directions to tackle these issues are proposed in [1].

Closer integration between EM and conventional programs: Experience of program development from EM models indicates that there is potentially considerable information loss in the transition from system concept to programmed system [1]. For instance, specialisation and efficiency is typically obtained through optimisations that restrict functionality and obscure the empirical roots of the system conception. Relevant topics for future research include potential ways of integrating ISMs with programs (cf. the way in which engineers attach monitoring instruments to an executing system) and the development of empirical techniques for bottom-up development of ISMs.

Acknowledgments

We are indebted to the members of the Empirical Modelling Research Group, and to many undergraduate students whose project work has contributed to this paper.

References

1. J Allderidge, W M Beynon, R I Cartwright, Y P Yung *Enabling Technologies for Empirical Modelling in Graphics* CS-RR#329, University of Warwick 1997
2. W M Beynon, M D Slade, Y W Yung *Parallel Computation in Definitive Models* CONPAR 88, BCS Workshop Series CUP, 1989, 359-367
3. W M Beynon, I Bridge, Y P Yung *Agent-oriented Modelling for a Vehicle Cruise Control System* Proc ESDA'92, ASME PD-Vol.47-4, 1992, 159-165
4. W M Beynon, M T Norris, S B Russ, M D Slade, Y P Yung, Y W Yung *Software Construction using Definitions: an Illustrative Example* CS-RR#147, University of Warwick 1989
5. W M Beynon *Programming Principles for the Semantics of the Semantics of Programs* CS-RR#205, Univ of Warwick 1992
6. W M Beynon *Agent-oriented Modelling and the Explanation of Behaviour* in Proc Int Workshop on Shape Modelling, Parallelism, Interactivity and Applications, Dept of Computer Software TR94-1-040, Univ of Aizu, Japan 1994, 54-63
7. W M Beynon, Y P Yung, A J Cartwright, P J Horgan *Scientific Visualisation: Experiments and Observations* Proc Eurographics Workshop on Visualization in Scientific Computing, 1992, 157-173
8. W M Beynon, S B Russ *Empirical Modelling for Requirements* CS-RR#277, University of Warwick 1994
9. W M Beynon *Modelling State in Mind and Machine* CS-RR#337, University of Warwick 1998
10. W M Beynon, M Farkas, Y P Yung *Agent-oriented Modelling for a Billiards Simulation* CS-RR#260, Univ of Warwick 1993
11. F P Brooks *The Mythical Man-Month Revisited* Addison-Wesley 1995
12. R Brooks *Towards a Theory of Comprehension of Computer Programs* Int'l J. Man-Machine Studies Vol 18, 1983, 543-554
13. G Cornell *Core JAVA* SunSoft 1996
14. M S Deutsch Focusing Real-Time Systems Analysis on User Operations IEEE Software, Sept 1988, 39-50
15. J Good, P Brna *Explaining Programs: when talking to your mother can make you look smarter* Proc PPIG-10 Annual Workshop, 61-69
16. D Harel *Biting the Silver Bullet: Towards a Brighter Future for System Development* IEEE Computer, Jan 1992, 8-20
17. W Kintsch, T van Dijk *Toward a model of text comprehension and production* Psychological Review, 85, 363-394
18. A van Mayrhauser, A M Vans *Program Comprehension During Software Maintenance and Evolution* IEEE Computer, August 1995, 44-54
19. N Pennington *Comprehension strategies in programming* Empirical Studies of Programmers: Second Workshop, New Jersey, Ablex Publishing Co. 1987, 100-113
20. A Pnueli *Applications of Temporal Logic to Specification and Verification: A Survey of Current Trends* Lecture Notes in Computer Science #224, Springer-Verlag, 1986, 510-584
21. Brian Smith *Two Lessons in Logic* Computer Intelligence Vol 3, 1987, 214-218

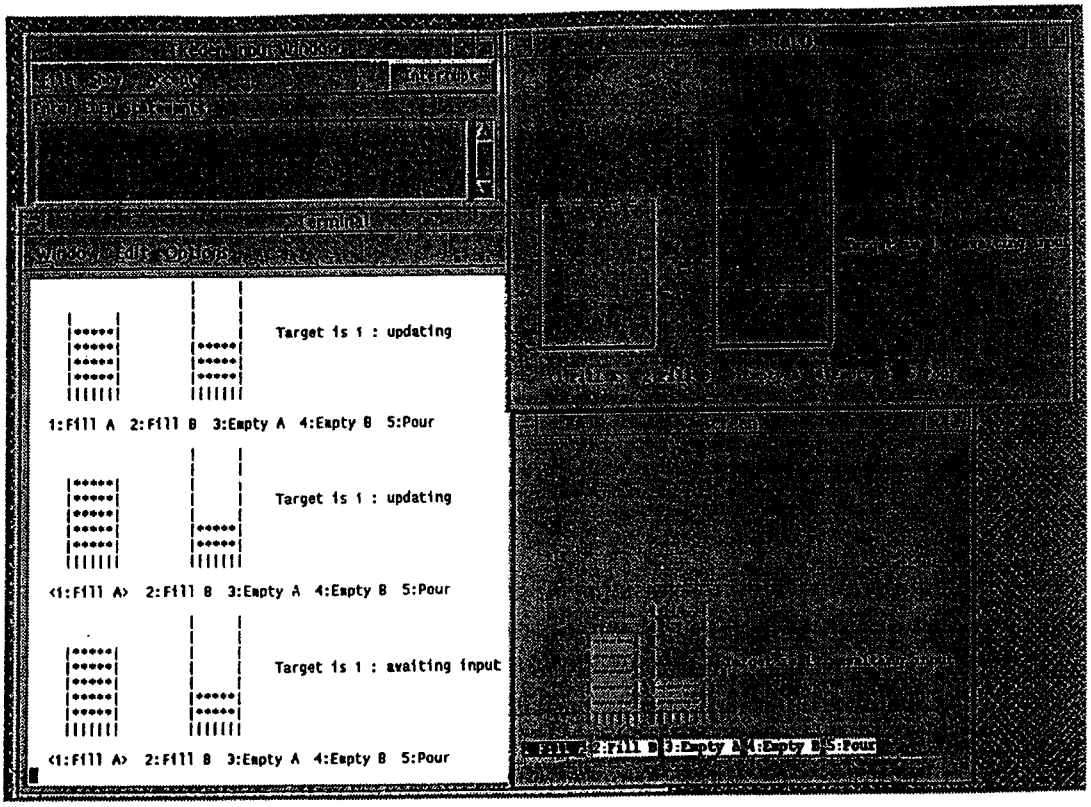

Figure 1

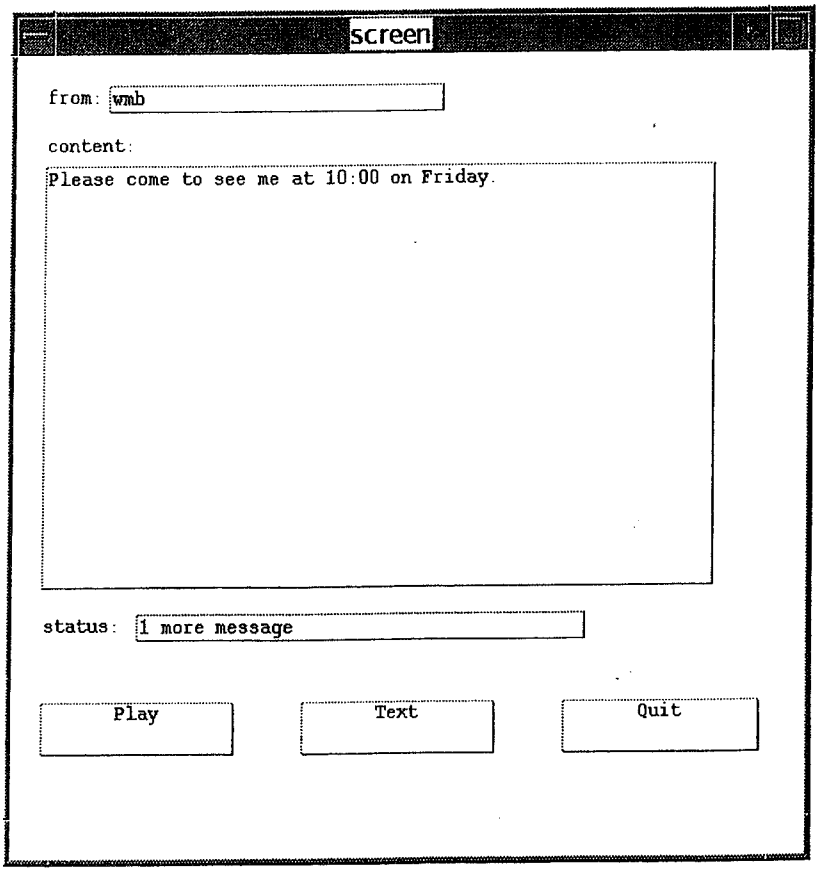

Figure 2